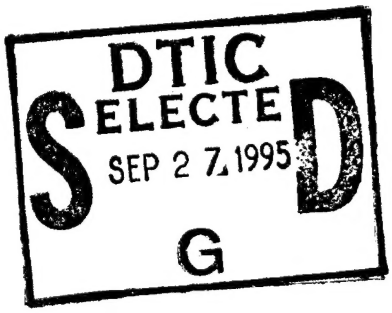
A Design Study for a New Pattern Recognition Accelerator

Interim Technical Report
Contract No. N00014-94-C-0164

Prepared by
AT&T Bell Laboratories
January 27, 1995



Abstract

The design of a programmable high-performance processor VLSI device is considered for a wide variety of image signal processing applications, including the potential for high-speed pattern recognition (e. g., text character recognition), image analysis, graphics rendering, speech recognition, and other computationally intensive algorithms. The approach is described in detail and its performance is analyzed in the context of two computationally demanding algorithms: linear filtering (e. g., two-dimensional discrete cosine tranform), and neural network evaluation for character recognition. Subject to verification by simulation, the design described meets the benchmark objectives of the study.

Table of Contents

1. Introduction

1.1. Goal

The objective of this project is to study the design of a programmable high-performance processor chip capable of supporting a wide variety of image signal processing applications. Potential computationally intensive applications enabled by such a chip include high-speed pattern recognition (e.g. OCR), image analysis, graphic rendering, and speech processing.

The main performance goals of this study are high-speed linear filtering (e. g. DCT) and image character recognition (> 5,000 characters per second) with one processor chip. These tasks are computationally very demanding and serve as a *benchmark* for our project. If they can be programmed successfully on this processor chip, it has enough computational power for many other image signal processing applications. It is critical, however, that the computation is carried out with a homogeneous array of *general-purpose* processing elements. This approach is preferred over the use of specialized units that may provide the required computational power for one particular application, such as image decompression or compression, but are unusable for other processing steps that may be required for other applications. In the following, we refer to this processor chip as the *Homogeneous Image Processor* (HIP).

1.2. Motivation and Vision

With current 0.5 μm technology, the stated goal is very hard to achieve and necessarily results in a large, expensive, and power hungry chip. However, as silicon technology shrinks down to 0.35 μm and 0.25 μm such processors will become small, affordable, and low in power consumption. We expect this to be achievable within the next two years. Then, homogeneous multi-processor chips will replace the currently used special-purpose chips in many cases. This is because a homogeneous multi-processor can be used not only for one, but for many computationally intensive applications including image analysis, pattern recognition, graphics rendering, and speech recognition. Table 1.1 shows how a chip's computational power increases as the minimum feature size shrinks. For our chip design study we assume a 0.35 μm CMOS technology.

This development is a logical continuation of the processor evolution experienced so far. Figure 1.1 graphically represents this evolution which is driven by the *increasing chip integration densities* (or equivalently, decreasing feature sizes) and the desire for more performance (faster computation, new medias). The first RISC processors (e.g. MIPS R2000) contained not much more than an integer execution unit. Floating point accelerators (e.g. MIPS R2010) were available for scientific applications as separate chips operating as a co-processor. As integration densities increased, it became common to *integrate* the floating point unit (FPU) onto the main processor's chip (e.g. MIPS R4000 series).

Digital signal processors (DSP) are processors which are optimized for modem and audio applications. Many of today's PCs are equipped with a DSP for modem functions and sound effects such as voice input/output. The newest development, which started with Apple's PowerMac series based on the IBM/Motorola PowerPC processor, is to get rid of the DSP chip and *integrate* the DSP functionality onto the main processor.

This is a typical pattern: New technologies start out as separate add-on chips which will eventually make their way onto the main processor chip. We predict that the same

Technology	Density	Clock Speed	Computation
0.9 μm CMOS	1.0	1.0	1.0
0.5 μm CMOS	2.8	1.6	4.5
0.35 μm CMOS	5.0	2.3	11.5
0.25 μm CMOS	9.7	3.1	30.1

Table 1.1: Increase of computational power per chip (same chip area assumed).

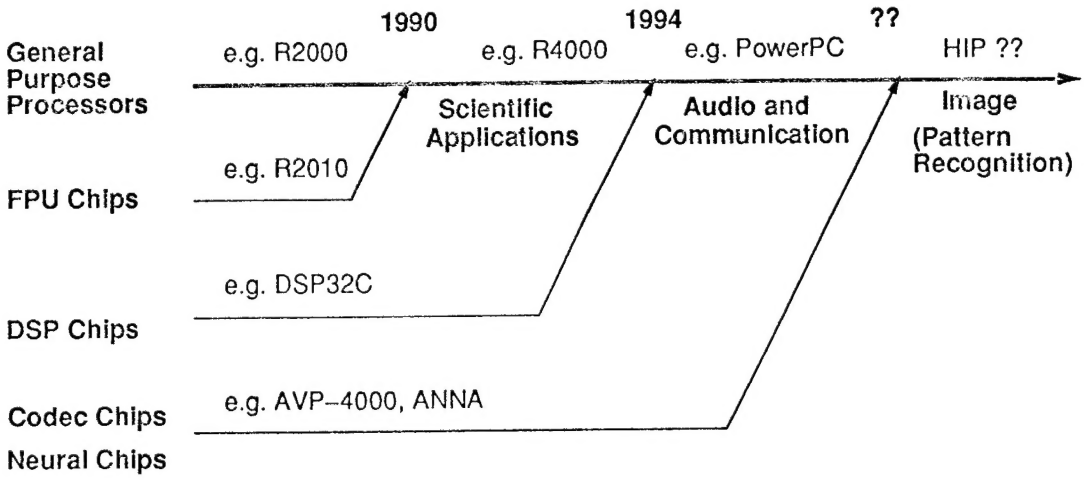


Figure 1.1: Evolution of the general purpose processor.

will happen to specialized pattern recognition chips (e.g. AT&T ANNA Artificial Neural Network Accelerator [2]). The Homogeneous Image Processor described in the following is our view of how this evolution will continue.

1.3. Approach

This section discusses the major design decisions related to the HIP chip project.

Instruction Set Architecture: Standard vs. Proprietary

The development of software tools for a new processor is a major effort. If the processor is based on a wide-spread instruction set architecture, existing tools such as assembler, compilers, and linkers can be reused and would not have to be developed from scratch.

For the HIP the SPARC V8 instruction set architecture was chosen for the following reasons: (i) a huge amount of software is available, (ii) AT&T's expertise in SPARC chip designs, (iii) the simplicity of the V8 instruction set (RISC), and (iii) the large register file (136 registers).

This approach is similar to that taken by the ICSI/Berkeley team for their Torrent T0 [3] which is based on the MIPS instruction set architecture. The Texas Instruments TMS 320C80 processor, on the other hand, is based on two proprietary instruction sets.

Vector Unit: Deep Pipeline vs. SIMD

Multimedia applications rely heavily on vector operations: Discrete Cosine Transform (DCT) and its inverse (IDCT) require the computation of many matrix products; pattern recognition and image analysis requires the computation of many convolutions; speech recognition requires the computation of many weighted Euclidean distances, and so on. It is therefore reasonable to enhance the SPARC core with a vector unit to speed up this type of computation.

Traditionally, vector units are built with deeply pipelined execution units (e.g. Cray Supercomputers). The deep pipelines allow these units to run at a very high clock frequency corresponding to a very high throughput. The long latencies caused by the deep pipelines can be tolerated in many vector computations because of the independency of successive operations. However, substantial performance losses are encountered on short vectors. The arithmetical operations are usually carried out on double-precision floating-point values since these computers are mostly used for scientific calculations (e.g. simulations in physics).

Another type of vector unit has been pioneered mainly by the neural-network community (chips such as ANNA [2], CNAPS [4], T0 [3]). In these designs, SIMD parallelism is used to speed up the vector computations. Like in the nervous system, the computational power is not (only) derived from a high-speed processing element but from (massive) parallelism. These vector units have a smaller latency (shallow pipeline), are more scalable (number of processing elements), and require less power[1]. The arithmetic operations are typically carried out on short fixed-point values since the image and sound data types processed by these systems require no more than 8 to 16 bits.

The vector unit for the HIP is of the SIMD type and is called Reduced precision Vector Unit (RVU) in the following.[2] Besides the SIMD architecture the RVU contains a number of innovations such as a programmable trade-off between precision and parallelism, a vector reduction, and a vector formatting unit which will be explained in a later section.

Parallelism: SIMD vs. MIMD

It has been mentioned before that the SIMD vector unit is scalable, i.e., its peak performance can be made arbitrarily large by adding more processing elements (PE). In addition, the SIMD processing elements are very cheap: only the arithmetic units (as opposed to the control units) must be replicated, furthermore the arithmetic units are small due to the reduced precision. Adaptive Solutions' CNAPS system exploits this fact by integrating 64 PEs (cascadable to 512 PEs) on a single chip.

Unfortunately, the *actual* performance (as opposed to the *peak* performance) does not scale as easily for many applications. The reason is that small vectors (with fewer components than PEs) may be prevalent in the code, or parts of the code may be non-vectorizable. Practice shows that the effectiveness of a SIMD system decreases, if more than 8 to 16 PEs are used. Instead of adding more PEs to the same control unit, it is more efficient to use several independent control units, thus switching to the more general MIMD architecture.

---

[1] If the supply voltage is reduced by n and the parallelism is increased by n the computational power stays the same, but the dynamic power dissipation is reduced by n [E. Vittoz, ISSCC94]

[2] This unit was originally named DIANA [5] for DIgital Artificial Neural network Accelerator. It was later renamed to RVU in order to signify its usefulness outside neural network applications.

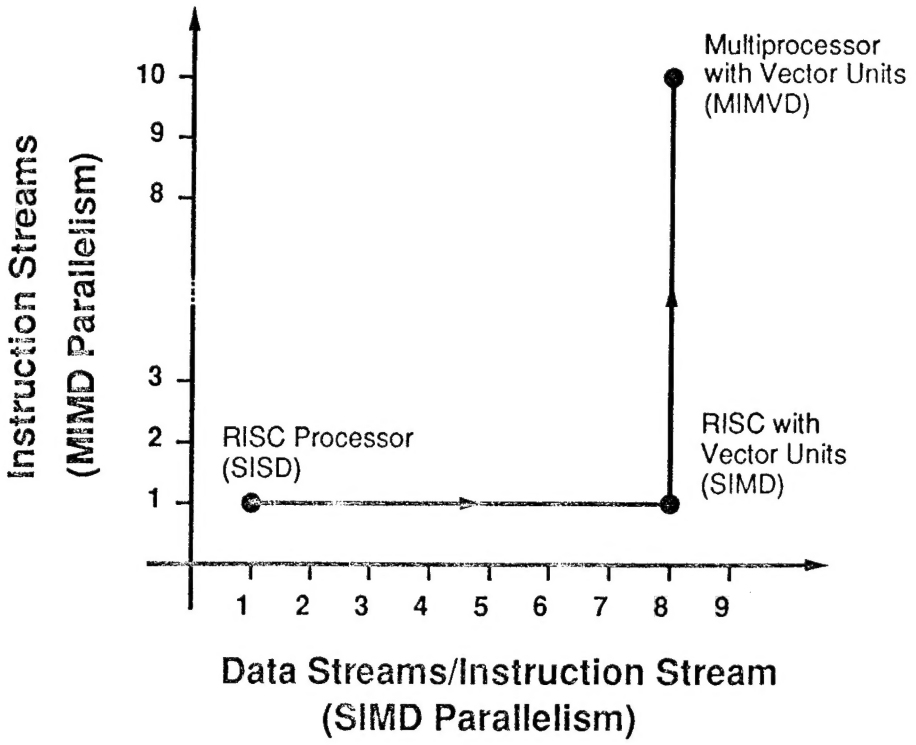


Figure 1.2: Options for parallelism: SIMD vs. MIMD.

Figure 1.2 illustrates this process graphically. As we follow the line in the direction of the arrow we go to more and more powerful parallel computers for multi-media applications while keeping the cost as low as possible. We start out with a single processor (1,1), then start to increase the SIMD parallelism (add a SIMD vector unit) because this is the cheapest way to get speed-up, eventually we are forced to move to MIMD parallelism because SIMD with many PEs becomes inefficient.

1.4. HIP Chip Architecture

Each block of the architecture will be discussed briefly in this section. A fuller discussion is provided in the subsequent chapters.

SPARC/RVU Processing Elements

At the heart of the HIP design is a homogeneous array of 10 processing elements each consisting of a SPARC core, a vector unit (RVU) and a local buffer memory. The SPARC core executes SPARC V8 integer operation at 100 MHz clock frequency and is implemented as a classical 5 stage RISC pipeline. The 4 KByte buffer memory is used as an instruction cache as well as a data buffer. The boundary between these two memory types is programmable. The buffer can for instance be configured for 2 KByte instruction cache and 2 KByte data buffer. The key innovation of this processing element is the RVU vector unit which delivers the computational horse power needed for multimedia applications.

Several recently announced processors, such as the Ultrasparc [6] and the TMS320C80, support so called split-word operations (e.g. four 8-bit adds in one instruction). The operations carried out by the RVU can be viewed as a generalization of these operations.

Figure 1.3 shows the RVU block diagram. The two source operands are *vectors* with eight 8-bit or four 16-bit components. The result can be either a scalar or one of the two vector formats. The RVU reads both operands and writes the result to the SPARC register file (register to register architecture).

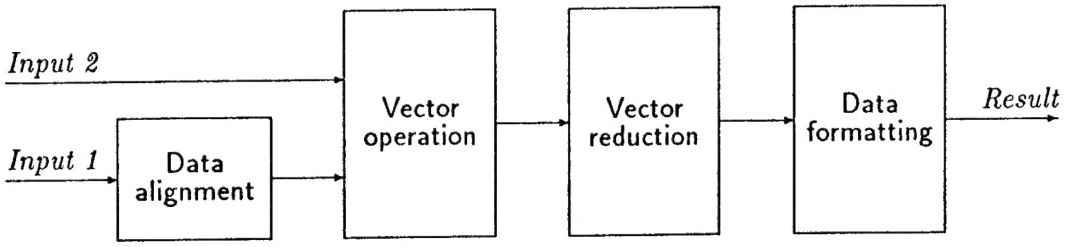


Figure 1.3: RVU block diagram

Data Aligning Block This block shifts one input vector (*Input 1*) against the other (*Input 2*) by a selected number of components for the case that the input vectors are not word aligned. This function is required in convolution operations. To assist the aligning process, previously read vectors are buffered locally. This block can also replicate or broadcast a scalar value into all components of the out-going vector. This function is needed for vector-scalar operation (e.g. multiply a vector by a scalar).

Vector Operation Block This block carries out component-wise arithmetic operations on its two aligned input vectors (so-called split-word operations). Typical operations are multiplication, addition, subtraction, and absolute difference. The result is a vector with more bits per component than the input vectors. This block is a single-instruction multiple-data (SIMD) parallel processor since the same operation is carried out on multiple data streams (the vector components).

Vector Reduction Block This block either passes the vector directly to its output (no operation) or sums up the vector components (reduction) with an adder tree. The scalar result is further added to one of eight 64-bit accumulators part of this block. The accumulators allow the efficient vector-to-scalar reduction of vectors with more than 4 or 8 components. The vector-to-scalar reduction operation is needed for instance to compute dot-products and Euclidean distances. Optionally, this block can also perform other reduction operations such as finding the minimum component to support the Viterbi algorithm.

Data Formatting Block This block formats the output vector (*Result*) to make it compatible with the input vectors such that vector operations can be chained together without intervening operations. If the output value from the vector reduction block is a vector (no reduction), all components are scaled (shifted right) by the amount given by one of four scale registers. Then the components are saturated to the selected byte or nibble format. If the output is a scalar (reduction selected), it is either output as a 64-bit value or scaled and saturated to the selected byte or word format. Optionally, several scalars can be packed into one word to form a vector. The latter feature is useful for vector-matrix multiplications and convolutions with multiple kernels.

		Alignment	Vec. Oper.	Vec. Red.	Format
Vector Addition		shift	add	–	scale
Vector Scaling		bcast	mult	–	scale
Manhattan Distance		shift	sub/abs	acc	scale
Euclidean Distance	1.	shift	sub	–	scale
	2.	–	mult	acc	scale
Weighted Euc. Dist.	1.	shift	sub	–	scale
	2.	–	mult	–	scale
	3.	–	mult	acc	scale
Dot Product		shift	mult	acc	scale
Vect.-Mat. Product,	1.	shift	mult	acc	scale/pack
Mat.-Mat. Product	2.	shift	mult	acc	scale/pack
	⋮				
Convolution	1.	shift	mult	acc	scale
	2.	shift	mult	acc	scale
	⋮				
Viterbi Step		shift	add	min	scale

Table 1.2: Common vector operations and their implementation on the RVU.

The individual blocks are controlled by independent subinstructions (orthogonal instruction set) encoded as part of the SPARC `CPop1` instruction. This gives the RVU a huge amount of flexibility. If, for instance, the vector operation block carries out a multiplication and vector reduction is applied, then the dot product of the two input vectors is computed (used in filtering and neural networks). Table 1.2 gives additional examples of how common vector operations can be synthesized with the 4 blocks of the RVU. Note that some operations need multiple cycles.

The RVU operates as a *co-processor* to the SPARC core. This means that the standard SPARC instruction set is extended with the RVU instructions. Because RVU instructions are executed in a separate pipeline, a SPARC and an RVU instruction can be issued at the same time. The SPARC/RVU processing element can operate in two modes: In one mode, only SPARC instructions are executed and in the other one, a SPARC and an RVU instruction are executed in parallel.

SPARC/FPU Processing Elements

This optional processing element is similar to the SPARC/RVU pair except that the vector unit has been replaced by a single-precision floating-point unit. This processing element is useful for example for graphic front-end computations (coordinate transformations) and other functions which require higher precision.

DMA Controller and External Memory

All processing elements, I/O and external memory systems are interconnected with a high-speed DMA (direct memory access) bus for data and instruction transfers. Besides

the DMA bus is a lower bandwidth control bus which is used by the SPARC cores to program the other units. For instance, a DMA request is submitted over this bus to the DMA controller.

The intelligent DMA controller (I-DMA) can transfer contiguous memory blocks as well as 2-d image patches and it can perform spatial subsampling while transferring the data. All memories (external memory and local buffers) are embedded in a single address space, however, only a subset of this space (the local buffer) is visible to each PE.

The External Memory is synchronous DRAM. Compared to normal DRAM it has about twice the data throughput but with a latency of several clock cycles. It is therefore ideal to communicate large data sets at high speed. The data width is planned to be twice the width of the internal DMA bus (2×64 bits). This allows to operate the synchronous DRAM at half the clock speed of the processor array.

Interprocessor Communication

The DMA controller allows point-to-point communication (the standard DMA operation) as well as *intelligent communication.* The latter means, that several sources (processors, memory, I/O) send different parts of a common data set and each information sink (processors, memory) makes a local copy of the interesting part of that data.

Intelligent communication has been proven to be very efficient in many data-parallel applications of the MUSIC [7] project. Compared to normal DMA it has the following advantages:

a) *Less data traffic.* Overlapping regions of search areas are communicated only once and many processors can access this information at the same time.

b) *Reduced latency.* The issuing of one communication transaction is enough to distribute the necessary data subsets to all processors, compared to an individual transaction for each processor in case of standard DMA transfer. The communication overhead therefore appears only once.

c) *Simple programming.* The data source does not have to consider to what location to send the data to. It just sends it out to the bus. The information consumers, on the other hand, do not have to consider where the data is coming from. All they need to know are the dimensions of the complete data set and what subset they are interested in.

A disadvantage of the intelligent communication appears only if the data subsets assigned to the processors do not overlap. In this case normal DMA transfer is more efficient.

Input/Output Interfaces

The chip communicates with the surrounding world through two I/O channels, a host interface, and auxiliary interfaces. The two I/O channels, one for input and one for output, generate (or synchronize to) the I/O timing signals. Data transfers to and from the frame memories (located in the external memory) are requested by the I/O interface and carried out by the DMA controller.

Through the host I/O interface, the host computer can configure and program the HIP (e.g. download code) and exchange data. Auxiliary I/O channels such as ISDN, I^2C support host-less systems.

1.5. Performance Discussion

In order to get a high sustained performance we need a good balance between computational performance, communication bandwidth and memory size. The following short discussion shows that the necessary conditions (e.g. peak performance) for DCT and for high speed character recognition are fulfilled. A much more detailed analysis (e.g. speed performance based on cycle count) will be given in the subsequent chapters.

Computational Performance

In case of linear filtering (DCT) usually a higher precision of 16 bits is required. In this case four multiply/add and four loads can be carried out in one instruction, corresponding to 12 RISC-like instructions. Table 1.3 shows the peak speed for each type of computation. MOPS stands for Mega Operations Per Second and R-MIPS stands for RISC-like Mega Instruction Per Second. Actual performance numbers can be found in Section 3.2.

Typical Application	Precision	MOPS	R-MIPS
Linear Filtering	16 bit	8,000	12,000
Non-vectorizable	32 bit	1,000	1,000

Table 1.3: Computational peak performance of 10 SPARC/RVU *processing elements.*

The most demanding operation is DCT/IDCT (Discrete Cosine Transform and its inverse). Real-time operations on images of TV resolution require about 1,200 MOPS. This requirement is well satisfied by HIP's computational power.

Communication Bandwidth

The communication bandwidth of the on-chip 64-bit DMA bus at a 100-MHz clock rate is 800 MByte/s. The bandwidth to the external memory is also 800 MByte/s (128-bit bus at 50 MHz).

Memory Capacity

The *on-chip buffer memory* of each processor is 4 KByte and is shared between the data buffer and the instruction cache. The size of the *external DRAM memory* is not critical and can be several 10 MBytes.

Chip Size and Pin Count

Assuming a 0.35 μm CMOS technology the SPARC/RVU processing element (including buffer and routing space) is estimated to be about 7.1 mm^2 in size. The SPARC/FPU processing element is slightly larger 7.5 mm^2. The proposed combination of 10 SPARC/RVU and 1 SPARC/FPU processing elements therefore fits comfortably on a chip with 100 mm^2 active area.

The synchronous DRAM memory interface requires about 150 pins; the host interface about 50 pins, the two I/O ports 44 pins, and the power supply about 100 pins. Therefore, a total of about 350 pins are needed. This pin count can be reduced significantly, if an advanced high-speed memory bus, such as RAMBUS, is used for the external memory.

1.6. Software Support

Software Development Tools

Because the original SPARC instruction set remains unchanged, all existing high-level language SPARC compiler can be used to produce code (e. g. `GNU C` and `C++`) and existing software can be reused. The RVU instructions are not part of high-level languages and can directly be accessed only from the assembly level.

The most efficient way to write software is to organize time critical code in small units, written in assembly language, and to write the rest of the program in the preferred high-level language.

The RVU assembly instructions are implemented as a set of macros for the standard SPARC assembler. See Section 2.6 for the instruction definitions.

Programming Paradigm

The HIP is a MIMD machine (Multiple Instructions Multiple Data) and each processing element could execute different code. The normal case however is that all processors execute the same program but work on different data subsets (*data-parallel*). This is called SPMD (Single Program Multiple Data). It has the advantage of being much easier to program than true MIMD, the load balancing is simpler and usually more efficient. Compared to a SIMD architectures (Single Instruction Multiple Data) no performance degradation occurs for conditional code execution.

Architecture Verification

A high-level software emulator for the SPARC cores has been written in `C++`. It is currently being extended to support RVU instructions, the interprocessor communication and the I/O units of the chip. It will be used to obtain accurate timing and performance estimations for important applications. Once the high-level architecture is proven correct, a more detailed model will be implemented in VHDL (VHSIC Hardware Description Language).

2. RVU Architecture

2.1. Overview

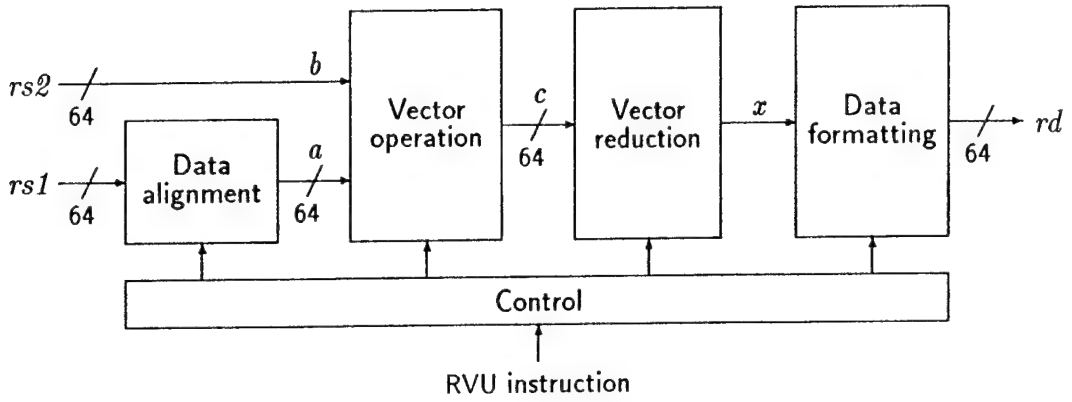


Figure 2.1: Block diagram of the reduced-precision vector unit (RVU).

Figure 2.1 shows the block diagram of the RVU which can be viewed as a generalization of split-word, vector instructions. The registers *rs1*, *rs2* (source) and *rd* (destination) are 64-bit registers from the RVU vector register file. The individual blocks perform the following operations:

Data alignment: Selects a 64-bit window from the combination of *rs1* and the EX-register (an extension register) or broadcasts the scalar in *rs1* to all components of the resulting 64-bit vector.

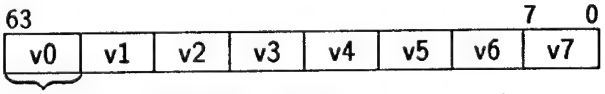
Vector operation: Carries out the following operations component wise: addition, subtraction, multiplication, absolute difference.

Vector reduction: Sums the vector components and accumulates the result in one of the accumulator registers. This block can be bypassed.

Data formatting: Scales and saturates the result and packs scalars into a vector (the results of several instructions can be packed into one vector).

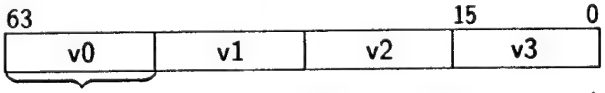
2.2. Data Formats

8×8-bit Vector:



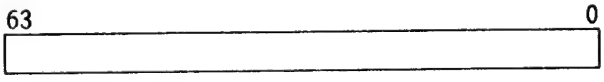
8 bit, 2's complement (range: −128...+127)

4×16-bit Vector:



16 bit, 2's complement (range: −32,768...+32,767)

64-bit Scalar:



destination only

2.3. Vector Registers

The General-Purpose Vector Register File (C[0]...C[15])

The RVU register file consists of 16 64-bit vector registers which are named C[0]...C[15]. They correspond to the first 16 locations of the SPARC Version 8 coprocessor register file. The SPARC `ldd` and `std` instructions can be used for data transfer operations from and to the memory (load and store operations).

The register C[0] has a "hardwired" value of zero. It always reads as zero, and writes to it have no effect.

The Accumulator Registers (AC[0]...AC[7])

There are eight accumulator registers named AC[0]...AC[7] corresponding to the SPARC coprocessor registers C[16]...C[23]. They can be accessed by the `ldd` and `std` instructions but they cannot be the source or destination of an RVU operation. The reduction unit uses them to accumulate the summed vector components.

The Extension and the Packing Register (EX, PK)

The extension register, EX, and the packing register, PK, correspond to the SPARC coprocessor registers C[24] and C[25]. They can be accessed by the `ldd` and `std` instructions but they cannot be the source or destination of an RVU operation.

The extension register is used as extension of *rs1* for data aligning. EX and *rs1* form a 128-bit word from which the data alignment unit picks a 64-bit window.

The packing register is used to assemble a vector from multiple scalars in the data formatting unit.

2.4. Control Registers

A total of 12 control registers control the operation modes of the RVU. They are mapped to the topmost SPARC ancillary state registers (ASR[20]...ASR[31]). The other ASRs ore either reserved or used by the SPARC core itself.

SF[j]	6 bit	(j = 0...7) A $\log_2$ scaling factor applied to the result in the data formatting unit (scalar or vector). Corresponds to ASR[20]...ASR[27].
WD	5 bit	Offset in multiples of the vector size applied to the [EX;*rs1*] register pair for the selection of a 64-bit window (if WD = 0 then a = EX). Corresponds to ASR[28].
SH	4 bit	Amount of shift applied to WD after the operation. Corresponds to ASR[29].
I	3 bit	Index register for the accumulator registers (AC[i]) in RVU operations. Corresponds to ASR[30].
J	3 bit	Index register for the scaling factors (SF[j]) in RVU operations. Corresponds to ASR[31].

2.5. Instruction Definition

The SPARC coprocessor operations (cpop1 and cpop2) are used to code RVU instructions.

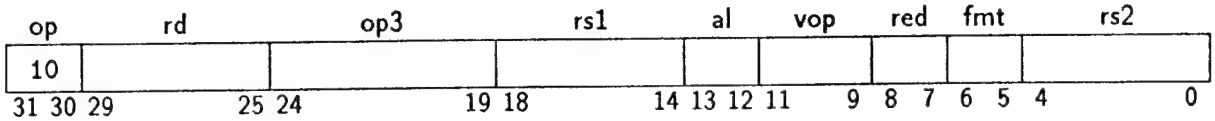


Figure 2.2: SPARC instruction word for RVU code.

op: SPARC instruction format for arithmetical, logicalal, shift, and coprocessor operations.

op3: Determins the data format (d = number of vector components, p = precision in bits).

110110: 8×8 bit words ($d = 8$, $p = 8$)
110111: 4×16 bit words ($d = 4$, $p = 16$)

These two **op3** codes correspond to the SPARC coprocessor operations cpop1 and cpop2.

rd: Specifies the destination register from the vector register file (0...15).

rs1: Specifies the source register 1 from the vector register file (0...15).

rs2: Specifies the source register 2 from the vector register file (0...15).

al: Data alignment:

00: $a = [rs1;rs1;rs1\ldots]$ (broadcast a scalar)
01: $a = [\mathrm{EX};rs1] \gg (64 - p \cdot \mathrm{WD})$ ($\mathrm{WD} = 0 \Rightarrow a = \mathrm{EX}$)
10: $a = [\mathrm{EX};rs1] \gg (64 - p \cdot \mathrm{WD})$; $\mathrm{EX} = rs1$
11: $a = [\mathrm{EX};rs1] \gg (64 - p \cdot \mathrm{WD})$; $\mathrm{WD} = (\mathrm{WD}+\mathrm{SH}) \bmod 8$;
when passing a modulo-d border: $\mathrm{EX} = rs1$.

vop: Vector operation:

000: $c_k = a_k \cdot b_k$ ($k = 0 \ldots \mathrm{d} - 1$)
001: $c_k = a_k + b_k$
010: $c_k = a_k - b_k$
011: $c_k = b_k - a_k$
100: $c_k = |a_k - b_k|$ (absolute difference)
101: reset all accumulator registers (AC[0]...AC[7])

The remaining codewords are not used.

red: Vector reduction:

00: $x_k = c_k$ (no vector reduction)
01: $x = \mathrm{AC[I]} = \mathrm{AC[I]} + \sum_k c_k$ ($k = 0 \ldots \mathrm{d} - 1$)
10: $x = \mathrm{AC[I]} = \mathrm{AC[I]} + \sum_k c_k$; $\mathrm{I} = \mathrm{I} + 1$
10: $x = \mathrm{AC[I]} = \mathrm{AC[I]} + \sum_k c_k$; $\mathrm{I} = 0$

fmt: Data formatting:

00:	$rd_k = \text{sat}_p(x_k \gg \text{SF[J]})$	if red = 0
	$rd = x \gg \text{SF[J]}$	if red $\neq$ 0
01:	$rd_k = \text{sat}_p(x_k \gg \text{SF[J]})$	if red = 0
	$rd = \text{PK} = (\text{PK} \ll p) + \text{sat}_p(x \gg \text{SF[J]})$	if red $\neq$ 0
10:	ditto; J = J + 1	
11:	ditto; J = 0	

2.6. Assembly Syntax

RVU assembly instructions are implemented by macro definitions for the Sun SPARC Version 8 assembler **as**. Note that a Version 8 assembler is mandatory (most Sun installations have a Version 7 assembler by default). The macros are loaded with the following command:

```
#include "rvumnem.h"
```

The assembler then has to be invoked with the `-P` option in order to process these macros correctly with `cpp`. To assemble a file named `demo.s` you would type

```
as -P -o demo.o demo.s
```

RVU Instructions

The general syntax for an RVU assembly instruction is

```
<format> <alignment> <operation> <reduction> <formatting> <operands>
```

The individual fields are macros, defined in `rvumnem.h`. The first field (`rvu_b8` or `rvu_h4`) expands to "`.word ...`" and each following field just adds the corresponding code bits. This works because the RVU instruction word is orthogonal. The meaning of the individual fields is described in Table 2.1.

There is one synthetic instruction, `rvu_nop` (for no operation) which expands to "`rvu_b8 wo vmul nr np reg(C0,C0,C0)`".

Load, Store and Move Instructions

The RVU *vector registers* (see Table 2.2) are implemented as SPARC coprocessor registers. Load and store operation with these registers are therefore performed by the SPARC LDDC and STDC instructions (`ldd` and `std`). The conversion of RVU vector register names to SPARC coprocessor register names is done by macros defined in `rvumnem.h`. Note that the general purpose vector registers (C[0]... C[15]), when used as operands in an RVU instruction, have a different mnemonic (capital letters, no %) than when used with SPARC load and store instructions (lower case letters with leading %). This is necessary to avoid naming conflicts in `rvumnem.h`.

The RVU *control registers* (see Table 2.3) are implemented as SPARC ancillary state registers. Move operations with these registers are therefore performed by the SPARC RDASR (`rd`) and WRASR (`wr`) instructions or by the synthetic instruction `mov`. The conversion of RVU control register names to SPARC ancillary state register names is done by macros defined in `rvumnem.h`.

Data format:	`rvu_b8`	8×8 bits (8 bytes)
	`rvu_h4`	4×16 bits (4 halfwords)
Alignment:	`bc`	broadcast
	`wo`	apply window only
	`wx`	apply window; exchange EX and *rs1*
	`ws`	apply window; shift window
Operation:	`vadd`	$c_k = a_k + b_k$
	`vsub`	$c_k = a_k - b_k$
	`vsun`	$c_k = b_k - a_k$
	`vsua`	$c_k = \|a_k - b_k\|$
	`vmul`	$c_k = a_k \cdot b_k$
	`clra`	reset AC[0]...AC[7]
Reduction:	`nr`	no vector reduction
	`sv`	sum vectors
	`si`	sum vectors and increment register I
	`sz`	sum vectors and zero register I
Formatting:	`np`	only scaling, no packing
	`ps`	pack and scale
	`pi`	pack, scale and increment register J
	`pz`	pack, scale and zero register J
Operands:	`reg(`*rs1*`,`*rs2*`,`*rd*`)`	any register (in capital letters without `%`, e. g. `C2`) from C[0]...C[15]

Table 2.1: RVU operation mnemonic

Register	Mnemonic	bits	Description
C[0]	`%c0/C0`	64	general purpose vector register 0
⋮	⋮	⋮	⋮
C[15]	`%c7/C15`	64	general purpose vector register 15
AC[0]	`%ac0`	64	accumulator 0
⋮	⋮	⋮	⋮
AC[7]	`%ac7`	64	accumulator 7
EX	`%ex`	64	extension register for *rs1*
PK	`%pk`	64	packing register

Table 2.2: RVU vector register names

Register	Mnemonic	bits	Description
SF[0]	`%sf0`	6	$\log_2$ scaling factor 0
⋮	⋮	⋮	⋮
SF[7]	`%sf7`	6	$\log_2$ scaling factor 7
WD	`%wd`	4	window offset for [EX;*rs1*] register pair
SH	`%sh`	4	shift of window offset after operation
I	`%i`	3	index register for accumulators
J	`%j`	3	index register for scaling factors

Table 2.3: RVU control register names

An Example Program

The following example program computes a 4×4 matrix/vector product with 16-bit precision per component. It is assumed that %i2 points to the 4-component vector and %i1 points to the matrix.

```
dot:  mov      4,%wd                          ! set window offset
      mov      %g0,%sf0                       ! scaling factor = 1
      ldd      [%i2],%c2                      ! load vector into %c2
      rvu_h4 wo clra sz pz reg(C0,C0,C0)      ! clear AC[0...7], I, J
      ldd      [%i1],%c1                      ! load first matrix row
      rvu_h4 wo vmul si ps reg(C1,C2,C0)      !    and mpy with %c2
      ldd      [%i1+8],%c1                    ! second matrix row
      rvu_h4 wo vmul si ps reg(C1,C2,C0)
      ldd      [%i1+16],%c1                   ! third matrix row
      rvu_h4 wo vmul si ps reg(C1,C2,C0)
      ldd      [%i1+24],%c1                   ! fourth matrix row
      rvu_h4 wo vmul si ps reg(C1,C2,C3)      ! result vector is in %c3
```

2.7. Pipelines

The SPARC core and the RVU have separate pipelines (integer and vector pipeline) which can work in parallel. This is most efficiently used to perform load, store and loop control operations in parallel to vector operations.

2.8. Chip Area Estimations

Multipliers

The following evaluation formulas are from the AT&T "High-Speed HS900C CMOS Standard-Cell Library Data Book," from January 1992, page 7-42. The number of transistors is given by

$$23.5 \cdot N \cdot M + 44.5 \cdot N - 3 \cdot M - 379 \tag{2.1}$$

where N and M determine the size of the input words. In case of four 16×16 multipliers the number of transistors totals to 25,204. The chip area required for 0.9 μm CMOS is

$$\text{height} = 25.7 \cdot M + 56.7 \tag{2.2}$$

$$\text{width} = 39.0 \cdot N \tag{2.3}$$

For the four 16×16 multipliers the total chip area is 1.67 mm^2 or 0.45 mm^2 if scaled to 0.35 μm CMOS.

By assuming that the adder tree in the reduction unit has about the same size as one multiplier and the total chip area for the RVU is about twice the size of the multipliers and the adder tree together, we can estimate the chip area of the RVU to be about 1.2 mm^2 for 0.35 μm CMOS.

Total Area

The following estimates, except for the RVU, are based on previous design experiences and are scaled from 0.5 down to 0.35 μm CMOS.

SPARC core:	1.8 mm^2
4K buffer:	3.2 mm^2
Busses (128 bit):	0.6 mm^2
Vector register file:	0.3 mm^2
Total SPARC:	5.9 mm^2
FPU:	1.6 mm^2
RVU:	1.2 mm^2

If an active chip area of 100 mm^2 is allowed then a possible distribution could be

10	SPARCs with RVU:	71.0 mm^2
1	SPARC with FPU:	7.5 mm^2
	Rest:	21.5 mm^2
		100 mm^2

3. Speed Performance Estimations

3.1. Peak Performance

Assuming a clock rate of 100 MHz (a conservative assumption) the configuration from Section 2.8 has a peak performance of 24 Gops (billions of RISC like operations per second) with 8 bits of precision and 8 Gops for linear transformations with 16 bits of precision. One floating-point unit is added for pre- and post processing, as well as for control tasks which require floating-point.

Table 3.1 compares the peak performance in million instructions per second (MIPS) and million operations per second (MOPS) between an array of SPARC cores with and without RVUs. A SPARC with an RVU is able to do a subtract, take the absolute value,

10 SPARCs:	without RVU	with RVU
R-MIPS	1,000	32,000
MOPS	1,000	24,000

Table 3.1: Peak performance of 10 SPARC processors with and without RVUs. R-MIPS means RISC-like MIPS.

do an add and a load on 8 vector components in parallel. The SPARC core by itself would need 32 instructions to perform the same task. The peak RISC-like MIPS rate (R-MIPS) of the RVUs is therefore 32 times higher compared to the SPARCs in Table 3.1.

3.2. Case Studies

The following performance estimations are based on cycle counts of corresponding assembler programs. They represent the peak processing performance of 10 SPARC cores with RVUs and a 100-MHz clock rate. Communication between the SPARCs is not yet considered.

Application	Performance
2-D linear filtering:	5.0 Gops
Character recognition:	2.8 Gops

Table 3.2: Performance estimation of three real applications.

1. **Two dimensional linear filtering** (DCT) on a 8×8 pixel block using 16 bit precision (no butterflies). This requires 406 ns corresponding to 5 Gops which is about 10 times the speed of the SPARC cores themselves. Assuming a 720×480, 4:1:1 frame resolution, the DCT could be computed on on 300 frames per second or 150 frames per second if also the inverse DCT is considered.

2. **Optical Character Recognition.** The first layer of LeNet-4 (a structured neural network with 264,580 connections for handwritten character recognition) has been coded. According to the cycle count the performance of a complete LeNet-4 was estimated to be 1.38 GCPS (billion connections per second) or 5,200 characters per second.

These results are summarized in Table 3.2.

Bibliography

[1] Bryan Ackland. The role of VLSI in multimedia. *IEEE J. Solid-State Circuits*, 29(4):381–388, April 1994.

[2] Bernhard Boser, Eduard Säckinger, Jane Bromley, Yann LeCun, and Lawrence D. Jackel. An analog neural network processor with programmable network topology. *IEEE J. Solid-State Circuits*, 26(12):2017–2025, December 1991.

[3] Krste Asanović, James Beck, Tim Callahan, Jerry Feldman, Bertrand Irissou, Brian Kingsbury, Phil Kohn, John Lazzaro, Nelson Morgan, David Stoutamire, and John Wawrzynek. CNS-1 architecture specification – a connectionist network supercomputer. Technical Report TR-93-021, University of California, Berkeley and the International Computer Science Institute, April 1993.

[4] Matthew Griffin, Gary Tahara, Kurt Knorpp, Ray Pinkham, and Bob Riley. An 11-million transistor neural network execution engine. In *ISSCC Dig. Tech. Papers*, pages 180–181. IEEE Int. Solid-State Circuits Conference, 1991.

[5] Eduard Säckinger, Yann LeCun, and Enrico Bocchieri. DIANA: A processor enhancement for multimedia algorithms. Technical Report TM 11359-940124-03, AT&T Bell Laboratories, 1994.

[6] Linley Gwennap. Ultrasparc unleashes SPARC performance. *Microprocessor Report*, pages 1–9, October 1994.

[7] Urs A. Müller, Bernhard Baumle, Peter Kohler, Anton Gunzinger, and Walter Guggenbühl. Achieving supercomputer performance for neural net simulation with an array of digital signal processors. *IEEE Micro*, 12:55–65, October 1992.